RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Expériences autour d'une nouvelle approche de conception d'un gestionnaire de travaux pour grappe

Nicolas Capit, Georges Da Costa, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, Olivier Richard *

Laboratoire ID-IMAG (UMR5132)/ Projet APACHE (CNRS/INPG/INRIA/UJF) Grenoble
{Prénom.Nom}@imag.fr

Résumé
Dans cet article nous présentons les choix de conception et l'évaluation d'un gestionnaire de travaux pour grappe de grande taille, baptisé OAR. Ce gestionnaire repose sur une conception originale qui réduit la complexité logicielle, permet une extension aisée ainsi qu'une bonne réponse au problème du passage à l'échelle. L'architecture globale repose principalement sur deux composants de haut niveau : un outil générique d'administration d'application (lancement, déploiement) passant à l'échelle et l'utilisation d'une base de données comme seul médium d'échange d'information entre les modules internes. Dans l'évaluation nous montrons à la fois le bon niveau de performance du système et sa capacité d'extension dans une utilisation de type *Global Computing* (utilisation des ressources inutilisées).

Mots-clés : Gestionnaire de ressources, *Batch Scheduler*, exploitation de grappe, passage à l'échelle, conception de système.

1. Introduction

La popularité des grappes de PC et leur large diffusion augmentent les besoins en outils souples et robustes pour leur administration et leur exploitation. Le projet de distribution Linux CLIC [1] qui regroupe un ensemble d'outils pour grappes est un exemple de réponse. Parmi ces besoins, un élément central pour l'exploitation des grappes est la disposition d'un gestionnaire de ressources. Il permet l'allocation de ressources processeurs aux tâches séquentielles ou parallèles soumises par les utilisateurs puis leur lancement ainsi que leur contrôle pendant leur exécution.

Il existe un grand nombre de gestionnaire de travaux pour grappes et machines parallèles. Parmi les plus connus nous pouvons citer PBS/OpenPBS[5], LSF[22], NQS[6], LoadLeveler[15], Condor[16], Glunix[13]. Les études [14] et [7] répertorient les principaux systèmes ainsi que leurs fonctionnalités. Actuellement peu de travaux de recherches s'intéressent directement à ces systèmes, bien que des études récentes ont été menées sur le lancement et le déploiement d'applications à grande échelle qui sont des éléments important dans la conception de ces systèmes [12]. De plus, dans l'ensemble, ces systèmes sont d'une conception ancienne et ne répondent pas complètement à certaines situations récentes (passage à l'échelle, flexibilité, forte extensibilité, robustesse).

Dans la suite de cet article nous motivons l'étude d'un nouveau système de gestion de ressources pour grappe, baptisé OAR, et posons les objectifs recherchés. Dans la section 3 nous présentons les choix de conception puis l'architecture générale en section 4. Nous présentons ensuite l'évaluation de notre système d'une part en comparaison au système OpenPBS et d'autre part en présentant une extension simple pour une exploitation de type *Global Computing*. Finalement nous concluons sur quelques perspectives.

*Supporté par les ACI-GRID CIGRID et CGP2P, le projet RNTL CLIC et BULL S.A. / Projet LIPS

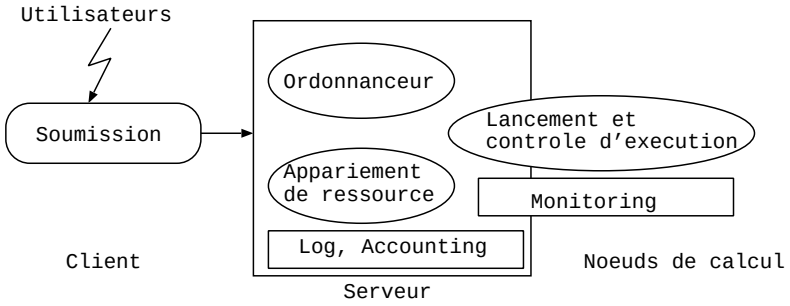

FIG. 1 – Architecture générale d'un gestionnaire de ressources pour grappe et machine parallèle.

2. Motivations, objectifs et systèmes apparentés

La figure 1 représente l'architecture classique d'un système gestionnaire de ressources. Les principales fonctions qui les composent sont : un module de soumission qui éventuellement exerce un contrôle sur la validité de la requête, un module d'ordonnancement associé à un mécanisme d'appariement (*matching*) de ressources et un module d'exécution qui contrôle le déroulement des tâches. A cela se rajoute des modules d'administration pour le suivi d'activité (*log*), de comptabilité (*accounting*), et de surveillance de ressources (*monitoring*).

Les principaux systèmes existants [14] ont été développés dans les années 90. Pour la plupart de ces systèmes, les concepteurs ont opté pour une intégration du plus grand nombre de fonctionnalités afin de répondre à un large ensemble de situations d'exploitations. L'accumulation de fonctionnalités a conduit à des logiciels de grande taille et d'une complexité importante (150 000 lignes de codes pour OpenPBS, cf paragraphe 5.1). Il s'ensuit que les niveaux de robustesse et de performance sont difficiles à maîtriser dans ces conditions. De plus, ce choix d'une architecture globalement monolithique rend toute extension imprévue ou toute modification potentiellement problématiques.

Le système PBS, par exemple, possède une API pour faciliter l'implantation de politique d'ordonnancement. Bien que cette API soit très riche en fonctionnalités elle n'est que très rarement exploitée. Ainsi, le projet Maui scheduler [2] qui a pour objectif le développement de politique d'ordonnancement avancée n'utilise le système PBS que comme système de soumission de tâches de type FIFO, laissant de coté l'API en question. Nous pouvons également remarquer que l'ordonnanceur Maui est devenu lui-même très important en taille de code. Ceci montre qu'il est difficile de maîtriser la complexité des logiciels de ce domaine.
Finalement la définition d'une API *universelle* pour le développement de modules d'ordonnancement reste un problème ouvert.
Une autre expérience basée sur le système PBS est celle du passage à l'échelle menée autour de la plateforme Cplant [8]. Une extension a été apportée sous la forme d'un module d'exécution et de contrôle fournissant une meilleure capacité de passage à l'échelle. La critique de cette expérience est similaire au cas précédent : l'extension se faisant par ajout de code et modification du système de base, la complexité globale du système est augmentée et la robustesse (voire la performance) de l'ensemble est fragilisée.

Récemment, une nouvelle génération de systèmes a fait son apparition, par exemple le système Maui Scheduling Molokini Edition [3]. Dans ce système, une base de donnée a été introduite pour maintenir certaines informations sur les utilisateurs et leur compte mais surtout comme solution de sauvegarde pour l'ensemble de l'état du système (ex. tâches en cours). Le même choix apparaît dans des systèmes apparentés comme XtremWeb [11], qui est un système pour l'exploitation des ressources inutilisées des machines volontaires (PC) connectées sur Internet. Par ailleurs, du coté des grappes de grande taille, sont apparus des systèmes de lancement et de contrôle d'exécution extensibles en nombre de noeuds (*scalable*) [12, 17, 9, 20]. Ils peuvent, pour certains, soit s'intégrer dans des gestionnaires existant, soit donner lieu au développement de nouveaux gestionnaire.

Ainsi le projet Storm [12] propose des fonctions de déploiement, de lancement et de contrôle d'application qui reposent sur des capacités de communication globale efficace fournies par un matériel réseau spécialisé.

L'exploitation d'une grappe de 225 PC, le projet *icluster*, a permis à notre laboratoire d'acquérir une expérience à la fois du point de vue des utilisateurs et des concepteurs de logiciels d'exploitation de ces plate-formes.
A coté de ces nouveaux systèmes et de l'évolution des plate-formes dans leurs tailles, nous pouvons remarquer une évolution des besoins des utilisateurs. Par exemple, en terme de réactivité à travers la soumission de tâches interactives lors de phase de développement. Ce besoin n'est pas récent mais des garanties sur sa satisfaction deviennent nécessaires. Un autre exemple de besoin nouveau est le support des applications multi-paramétriques. Celles-ci se caractérisent par un très grand nombre d'exécutions d'une même application avec un jeu de paramètres en entrée différent. Les problèmes posés ici sont liés à la gestion de l'ensemble des tâches à effectuer, à la maîtrise des erreurs (qui deviennent inévitables avec le nombre des tâches envisagées) et à la gestion des résultats (rapatriement, exploitation).

Du point de vue du concepteur de système, le principal besoin porte sur l'extensibilité du système. Les exemples pour lesquels une telle extensibilité est profitable sont nombreux : pour tester de nouvelles politiques d'ordonnancement, intégrer de nouveaux mécanismes d'administrations ou encore gérer des ressources dynamiques (qui se retrouvent dans les contextes d'exploitation de type *Global Computing* [16, 11] ou Cluster à la demande [10]). La difficulté d'obtention d'un système facile à étendre, réside dans la définition de modules et de composants parfaitement orthogonaux entre eux. Jusqu'à présent il ne semble pas exister de tel système de gestion de ressources et chaque extension nécessite d'importantes modifications.

Afin de répondre à ces besoins, nous avons défini les objectifs que devrait remplir un gestionnaire de ressources permettant à la fois d'offrir un service de gestion effectif et d'être une plate-forme de recherche. Les qualités prioritaires que nous avons considéré au cours du développement du système OAR sont la facilité d'extension, la capacité de passage à l'échelle et la robustesse. Pour atteindre cet objectif, nous avons choisi de réduire le nombre de fonctionnalités proposées par OAR par rapport à ses prédécesseurs, pour nous concentrer sur la majorité des besoins classiques des sites de production. Nous avons également conçu ce système avec en tête une architecture sous-jacente homogène, ce qui rend OAR moins adapté aux grappes hétérogènes (qui tendent à être moins répandues).

3. Choix de conception

Pour répondre à nos objectifs de simplicité, d'extensibilité et de robustesse, nous avons choisi de faire reposer la conception du système sur plusieurs composants logiciels de haut niveau : une base de donnée relationnelle (*MySQL*[4]) et un outil générique d'administration pour grande plate-forme (*Taktuk*[19, 18, 17]).
Ce choix permet de résoudre en partie le problème de la robustesse. En effet la base de donnée utilisée est largement reconnue pour sa stabilité. De plus la charge prévue se trouve être modeste (cf. chapitre 5.3). Pour l'outil générique, l'expérience acquise est plus réduite, par contre un nombre réduit de fonctionnalités sont utilisées et compte parmi les plus éprouvées. Finalement nous jugeons que les risques de dysfonctionnement des fonctionnalités réalisées par ces outils sont très faibles, ce qui permet de disposer d'une base logicielle solide pour le reste du système.
L'originalité du choix d'une base de donnée dans ce contexte se situe dans la place centrale qu'elle occupe au sein de l'architecture du système. En effet, contrairement aux systèmes Maui Molokini[3] ou XtremWeb[11] qui sont de conception classique, nous avons choisi d'ouvrir pleinement l'état interne du système en le maintenant entièrement dans la base de données et en utilisant celle-ci comme seul mode d'échange d'informations. La conséquence directe est qu'il n'y a pas de spécification d'API (*Application Program Interface*) pour les différents modules constituant le reste du système mais plutôt une spécification d'architecture de la base de données. Chaque module interagit dans le système via des requêtes SQL qui peuvent être locales ou distantes. Cette approche assure une modularité extrême à l'ensemble du système ainsi qu'une totale indépendance dans le choix du langage implémentant chaque

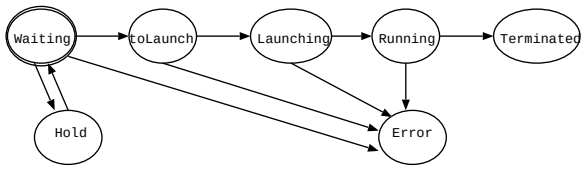

FIG. 2 – Diagramme d'état des tâches dans le système.

module (la plupart des langages de programmation possèdent des bibliothèques d'accès aux bases de données SQL). Grâce à cette approche, la spécification du système se réduit au diagramme d'état d'une tâche, à la signification des tables et de leurs champs et aux fonctions des différents modules. L'ensemble de ces caractéristiques assurent au système à la fois une grande simplicité et une forte extensibilité.
Dans une perspective de passage à l'échelle, notre choix de composants de haut niveau est une fois de plus un point clé. Pour l'échange de données au sein du système et la soumission massive de requêtes, nous pouvons nous reposer sur des systèmes de base de données prévues pour soutenir de fortes charges de travail. Pour le déploiement d'application parallèle et l'administration du système sur grappes massives (plusieurs milliers de noeuds), nous nous reposons sur le composant d'administration *Taktuk* spécifiquement conçu pour rester efficace sur un grand nombre de nœuds.
En ce qui concerne le développement des modules, nous avons retenu le langage de script interprété Perl. Ce langage possède nativement des structures de donnée de haut niveau comme les tables associatives et le support des expressions régulières. Ces capacités permettent un cycle de développement très court et renforcent la concision et la simplicité du code. Bien que n'ajoutant pas un surcoût démesuré, si le fait d'utiliser un langage de script pose un problème de performance, il est tout à fait possible et simple de réécrire un ou plusieurs modules dans un langage plus efficace (du moment qu'il dispose d'une bibliothèque d'interface avec une base de données SQL). Toutefois, ce choix de langage privilégie le critère de faible complexité logicielle par rapport à celui de la performance sans pour autant remettre en question la capacité de passage à l'échelle du système. Cette faible complexité est un élément favorisant à la fois la robustesse ainsi que l'extensibilité.

4. Architecture générale

Le système OAR est bâti selon le modèle présenté en section 2 et repris par la figure 1. Chaque élément fonctionnel de ce modèle est implémenté par un module indépendant. Ces modules communiquent par l'intermédiaire d'une base de données relationnelle et un module central est chargé d'orchestrer le fonctionnement de l'ensemble. Les utilisateurs communiquent avec le système par le biais de commandes de soumission, d'annulation, de monitoring, ... (à la manière de PBS).

Les spécifications nécessaires à la définition de l'architecture du système dans son ensemble sont le diagramme d'états des tâches et les différents tables et champs de la base de données. La figure 2 présente le diagramme d'état des tâches soumises au système. L'insertion, la retenue et la phase d'ordonnancement se déroulent dans les états *waiting* et *hold*. Les états suivants sont associés aux phases d'exécution. Un état de demande de retrait/arrêt de la tâche (non représenté) se positionne en parallèle, il est traité en priorité.

Nous ne détaillons pas l'ensemble des tables de la base de données. Deux de ces tables sont essentielles au cœur du système : la table décrivant les tâches, les ressources demandées ainsi que les dates des principaux événements (soumission, départ/fin d'exécution) et celle décrivant la plate-forme d'exécution ainsi que son état. Parmi les autres tables constituant le système, nous retrouvons celles stockant les informations pour l'ordonnancement, les règles d'admission et l'appariement de ressources.

4.1. Soumission de tâche

Du point de vue de l'utilisateur, le système OAR fonctionne à la manière de PBS : l'interaction avec le système s'effectue par l'intermédiaire de commandes indépendantes pour la soumission (commande *qsub*), l'annulation (commande *qdel*) ou l'observation de travaux (commande *qstat*). Ces commandes

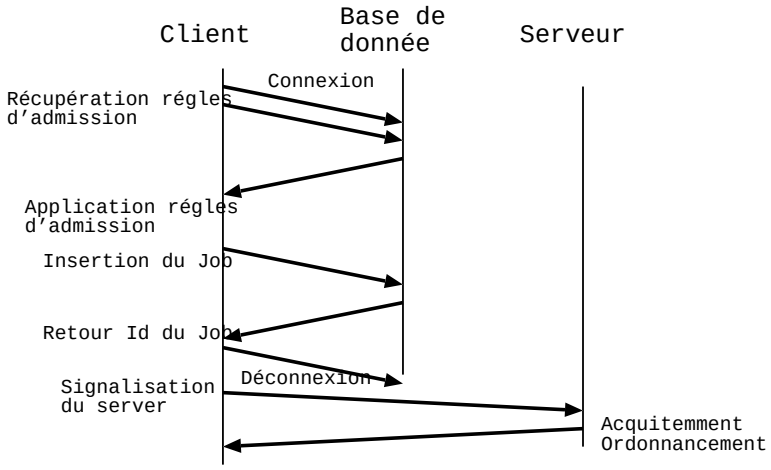

FIG. 3 – Déroulement de la soumission d'une tâche.

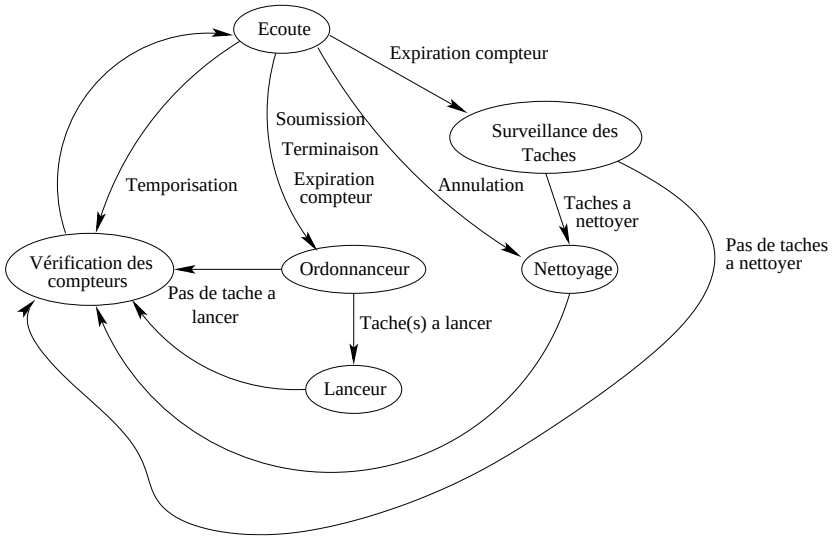

FIG. 4 – Automate du module central du système de gestion de ressource OAR.

sont découplées au maximum du reste du système, elles transmettent ou récupèrent les données dont elles ont besoin en accédant directement à la base de données et interagissent avec le système par l'intermédiaire de simples notifications envoyées à un module central.

La figure 3 présente le déroulement d'une soumission d'une tâche. Celle-ci débute par une connexion à la base de donnée suivie du rapatriement des règles d'admission. Ces règles servent à fixer les paramètres manquants (non fournis par l'utilisateur) liés à la soumission et à vérifier la validité de la requête. Parmi les paramètres pris en compte à la soumission nous retrouvons un identifiant de file d'attente, un temps limite, un contrôle des droits d'accès, etc. Les règles se présentent sous la forme de ligne de code en langage Perl et peuvent être remplacées par l'appel à un exécutable. La tâche est ensuite insérée dans la base qui, en retour, lui donne son identifiant correspondant à la clé nouvellement créée de l'index de la table des tâches. Après la déconnexion le module de soumission notifie le module central de l'arrivée d'une nouvelle tâche. Nous avons choisi d'implanter ce mécanisme de notification avec des `sockets` TCP.

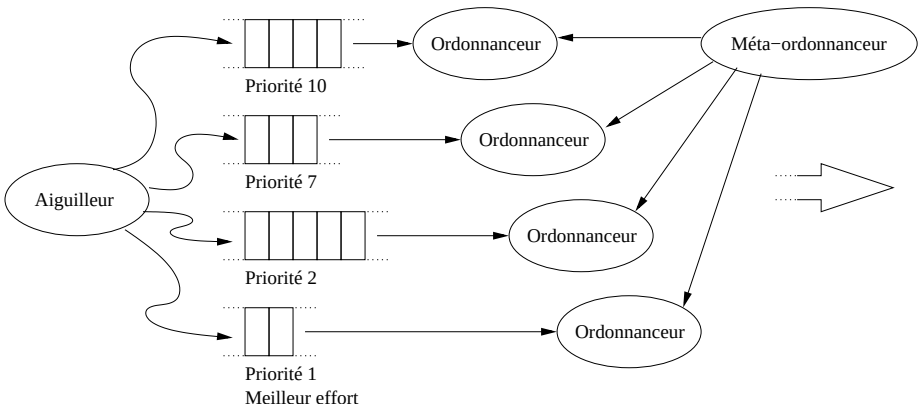

FIG. 5 – Structure d'un ordonnanceur dans OAR (exemple avec 4 files).

4.2. Module central

Le principal souci lors de la conception générale de OAR a été de garantir simultanément la réactivité et la robustesse du système. Ce sont ces objectifs qui ont guidé notre choix dans la structure du module central (figure 4). Ce module central est constamment à l'écoute de notification pouvant provenir de chacun des autres modules. C'est ce système de notification qui assure une bonne réactivité du système : dès qu'une commande ou un module ajoute des informations dans la base de données, le module central reçoit une notification et met en œuvre le traitement approprié. La robustesse, quand à elle est assurée par le découpage entre transmission des informations et notification. Pour qu'une requête soit traitée, il faut qu'elle transmette ses informations à la base de données et que le module de traitement correspondant soit lancé. Autrement dit, la centralisation des notifications ne constitue pas un goulot d'étranglement si le module central garantit le lancement régulier des modules de traitement : la tolérance du système à la charge de travail est déportée sur le système de bases de données. Comme nous pouvons le remarquer sur la figure 4, toute phase d'écoute des notifications dans le module central est immédiatement suivie d'une phase de traitement puis de mise-à-jour de compteurs internes afin, le cas échéant, de pouvoir lancer certaines opérations à intervalles réguliers. Dans le cas d'une surcharge du système, bien que l'afflux de notifications ne puisse pas être intégralement être pris en compte, le mécanisme des compteurs garantit un bon fonctionnement du système.

De façon plus détaillée, le module central est centré sur un état d'écoute (figure 4). De cet état d'écoute, toute nouvelle notification fait passer le module dans un état de traitement. Selon le cas il s'agit d'un ordonnancement suivi, si besoin d'un lancement, d'un nettoyage des tâches ne devant plus se trouver sur le système ou bien d'un contrôle de l'expiration du temps d'exécution alloué aux tâches. Entre deux phases de traitement, le passage par l'état de vérification des compteurs permet de réclamer à intervalle régulier le passage dans les états d'ordonnancement ou de contrôle de l'expiration du temps d'exécution alloué aux tâches.

4.3. Ordonnancement

A l'heure actuelle, le comportement du module d'ordonnancement est similaire à celui de PBS dans sa configuration par défaut (FIFO). Nous avons implanté l'appariement de ressources. Une requête peut demander des noeuds ayant des propriétés spécifiques, comme un commutateur réseaux, connectant la machine, particulier. Nous n'avons pas encore implémenté le remplissage des emplacements laissés vacants lorsque des tâches réclamant beaucoup de processeurs sont en attente (*backfilling*). En revanche, le support de tâches de type << meilleur effort >> (*Best Effort*) est déjà disponible.

La structure générale de l'ordonnanceur est décrite dans la figure 5. Lors de leur soumission, les tâches sont rattachées à une file d'attente. Chaque file d'attente est caractérisée par une priorité et possède son propre ordonnanceur. L'ordonnancement de l'ensemble est assuré par un module d'ordonnancement entre files d'attente (ou méta-ordonnanceur), qui détermine la file dans laquelle il faut chercher une tâche en attente

et lance l'ordonnanceur de cette file. Comparée à une approche centrée sur les tâches (ordonnancement d'un ensemble de tâches avec priorité, à l'image de Maui), cette approche perd une vision globale du problème dont pourrait tirer parti un ordonnanceur agressif (optimisant par exemple le rendement global par programmation linéaire). Cependant, ce découpage représente le meilleur compromis entre simplicité et richesse d'expression. D'une part, la conception et la compréhension de l'ordonnanceur deviennent extrêmement simple (politique de choix d'une file et politique de choix d'une tâche dans une file) et d'autre part, l'administration de l'ensemble du système est très intuitive (gestion des tâches au niveau des files : interruption/reprise d'une file, déplacement des tâches entre files, etc.).
A l'heure actuelle le méta-ordonnanceur est un simple ordonnanceur à base de priorités, il donne la main à l'ordonnanceur de la file non vide de plus haute priorité. De même, l'ordonnanceur en question est pour l'heure un simple algorithme FIFO (premier arrivé, premier servi). Cependant nous prenons en compte les tâches de type << meilleur effort >> . Ces tâches sont destinées à occuper les ressources inutilisées et sont interruptibles, sans reprise, si une autre tâche a besoin de ce noeud. Notre ordonnanceur, grâce à une modification de l'allocateur de ressources, est à même de déterminer un ensemble de tâches << meilleur effort >> à interrompre en cas de besoin (avec éventuellement une priorité entre de telles tâches). Cette fonctionnalité, requise dans les systèmes de calcul global (ex : Condor [16], XtremWeb [11]) nous a permis de démontrer la simplicité d'extension de OAR, y compris dans la prise en charge de fonction dépendant de plusieurs modules (cf. 5.1).

4.4. Un outil générique pour l'administration des grandes plate-formes

Habituellement dans les gestionnaires de ressources, les opérations de lancement, de déploiement et de surveillance sont effectuées par des processus démons spécifiques s'exécutant sur les noeuds de calcul. Dans le système OAR les modules ayant ce type de besoin repose sur l'utilisation de *Taktuk*[17, 19]. A l'origine, c'est un outil de déploiement d'applications parallèles pour grappes de grandes tailles (milliers de noeuds). Il peut être utilisé pour effectuer des opérations d'administration pour grappes en fournissant un service d'exécution distante parallèle et performant.
Il peut être utilisé de la même manière qu'un protocole d'exécution distante standard comme **ssh** ou **rsh**. Pour réaliser un lancement efficace sur les différents noeuds cibles, *Taktuk* est hautement parallélisé et distribué. Les différents appels d'exécution distantes sont réalisé avec des protocoles d'exécution distantes standards (rsh, ssh, rexec, etc...).
Pour une plus grande souplesse d'utilisation *Taktuk* est complètement indépendant des protocoles d'exécution distante utilisés. Pour cela, il emploie directement les clients standards associés à ces protocoles. Soit en utilisant ces commandes aux travers de nouveaux processus (<< fork-exec >> client), soit en utilisant des clients de protocole disponible sous forme de bibliothèques de fonctions.
Lors de la création de l'arbre de lancement une topologie fixe est déterminée et réalisée par la création de liaisons logiques (TCP/IP). Ceci permet d'établir un réseau virtuel qui est utilisé pour contrôler les applications déployées sur un ensemble de noeuds. Cette interconnexion permet aussi la redirection complète et bidirectionnelle des entrées/sorties des processus distants.

Le principe de parallélisation des différents appels distants utilise à la fois les propriétés de recouvrement (un noeud effectue plusieurs appels distants en pipeline) et de distribution des appels sur plusieurs noeuds. Tout nouveau noeud atteint participe au lancement. Ceci conduit à l'établissement d'un arbre de lancement couvrant l'ensemble des noeuds cibles.
En vue d'éviter toute surcharge des différents noeuds et pour permettre un lancement efficace passant à l'échelle sur des milliers de noeuds. *Taktuk* crée un ordonnancement dynamique de ces différents appels distants. Cet ordonnancement est basé sur un algorithme glouton de vol de travail, qui lui confère de bonnes propriétés d'adaptation à la charge du réseau et des différents noeuds cibles.

Les clients standards de protocole d'exécution distante possèdent leur propre mécanisme de détection de défaillance (noeud non atteignable). Cependant les mécanismes de détection utilisés par ces protocoles reposent sur des alarmes (temps limite d'attente de réponse) qui peuvent introduire une forte latence. Afin d'obtenir une meilleure réactivité et de limiter le temps d'un lancement sur un ensemble de noeuds, *Taktuk* peut ajouter son propre mécanisme de détection en bornant lui-même ces alarmes. Tout noeud non atteint à l'issue du temps d'alarme est considéré comme défaillant. Ce mécanisme permet d'offrir à

	OpenPBS	Maui Scheduler (+ OpenPBS)	Maui Scheduler Molokini	*Taktuk*	OAR (+*Taktuk*)
Version	2.3.16	3.2.5	1.5.2	3.0	-
Langage (majoritaire)	C	C	Java	C++	Perl
Nb fichiers sources	350	142	116	120	30
Nb lignes sources	148000	142000	25000	19500	4500

TAB. 1 – Éléments de comparaison logicielle pour différents gestionnaires de ressources .

l'utilisateur la qualité de service de son choix : il peut choisir un comportement très réactif (au risque de considérer à tort des noeuds comme défaillants) ou bien un comportement plus lent mais plus soucieux de l'état réel des noeuds (garantissant une forte confiance lorsqu'une défaillance est suspectée).

5. Evalutations

Pour démontrer la validité de notre approche nous avons effectué 3 types d'évaluations. Le premier est une comparaison qualitative de la complexité logicielle de plusieurs systèmes de gestion de ressources. Le deuxième porte sur les capacités d'extension du système OAR. Nous terminons par une comparaison des performances de OAR avec le système OpenPBS.

5.1. Complexité

Afin d'évaluer qualitativement la complexité logicielle des différents systèmes de gestion de ressources, nous nous fondons sur les indicateurs suivants : le langage majoritairement utilisé, le nombre de fichiers sources et le nombre total de lignes de code source. Dans les décomptes, nous avons seulement considéré les fichiers nécessaires au fonctionnement du système à l'exclusion de tous les fichiers relatifs à la documentation, aux interfaces graphiques, aux outils d'installation, etc. Le tableau 1 regroupe les données pour ces différents indicateurs dans les systèmes suivants : OpenPBS[5], Maui Scheduler [2], Maui Scheduler Molokini Edition [3], XtremWeb[11], OAR. Dans le cas des systèmes Maui Scheduler et OAR, les fonctions effectives d'allocation de ressources (pour Maui) et de monitoring (pour OAR) sont assurées respectivement par OpenPBS et *Taktuk*. Il est donc nécessaire d'ajouter la complexité supplémentaire due à ces outils pour obtenir la complexité de l'ensemble du système.

Ces résultats révèlent que les systèmes de première génération, que sont OpenPBS et Maui Scheduler, sont de loin plus complexes que Maui Molokini ou OAR. Ceci est dû d'une part au choix du langage d'implémentation du système et d'autre part au nombre de fonctionnalités prises en charge directement par celui-ci. Dans le cas d'OpenPBS, la complexité logicielle du système est d'autant plus importante que l'accent a été mis sur le nombre de fonctionnalités offertes par le système, la prise en compte de l'hétérogénéité et le choix du langage C pour l'implémentation de ces fonctions. A l'inverse, des systèmes comme Maui Scheduler Molokini ou OAR (+*Taktuk*) démontrent que la complexité d'un gestionnaire de ressource n'est pas intrinsèque à ce type d'applications mais dépend fortement des choix de conception.

Ces deux systèmes possèdent des tailles de codes comparables. Notons que dans le cas de OAR, le composant *Taktuk* représente la plus grande partie du code du système. Cependant, bien que *Taktuk* soit nécessaire au fonctionnement de OAR, il est développé indépendamment de ce dernier. Autrement dit sa complexité est restée complètement cachée lors du développement de OAR lui-même. La différence notable entre Maui et OAR vient du nombre d'*API* qu'il est nécessaires de manipuler pour modifier ou développer une nouvelle politique d'allocation dans Maui. Dans notre système ceci se fait simplement par la construction de requêtes à une base de donnée *SQL*. Ainsi OAR est tout à fait représentatif de la nouvelle génération de gestionnaires de ressources : il démontre qu'une conception extrêmement simple est possible pour ce type d'application à condition de tirer parti des outils logiciels de haut niveau désormais disponibles.

5.2. Extension : *Global* ou *Desktop computing*

De plus en plus de systèmes sont mis en oeuvre dans les Intranet d'entreprise, les grappes ou l'Internet afin d'exploiter les ressources matérielles durant leurs périodes d'inactivité, on parle alors de *Global* ou *Desktop computing*. Généralement, ceci se réalise par l'ajout d'un mécanisme de détection de ressources li-

bres qui déclenche lui-même la récupération et l'exécution de tâches [21][11]. Lorsque la ressource hôte est de nouveau réclamée pour son utilisation normale, elle est alors restituée (éventuellement au détriment d'une tâche de *Desktop computing* en cours de calcul, on parle alors de tâche << meilleur effort >>). Généralement, ces systèmes s'attachent à être le plus transparents possible. Cependant, dans le cas de l'exploitation d'une grappe, la transparence n'est pas nécessairement souhaitée : pour permettre une comptabilité des ressources consommées ou pour effectuer des choix d'ordonnancement plus adaptés, il est souvent nécessaire d'impliquer le système de gestion de ressources dans la mise en place du *Desktop computing*.

Cette extension a été implémentée dans OAR en ajoutant une propriété aux travaux de type << meilleur effort >> . Cette propriété est positionnée par l'aiguilleur, lors de l'insertion de la tâche dans la base (par exemple, en précisant lors de la soumission l'appartenance de la tâche à une file d'attente de type << meilleur effort >>). Il est également nécessaire d'être à même d'annuler les tâches << meilleur effort >> en cours d'exécution sur des nœuds occupés. Nous avons choisi d'implémenter la demande d'annulation des tâches dans les fonctions qui fournissent les noeuds disponibles aux ordonnanceurs et l'annulation elle-même dans le module générique de nettoyage lancé habituellement lors de l'expiration du temps alloué à une tâche. Ainsi, dans cette nouvelle version des fonctions de détermination des ressources libres, l'information doit remonter jusqu'au module central afin que celui-ci lance le module de nettoyage en question. L'inconvénient majeur de cette approche est que l'information doit transiter à travers divers modules du système. Ceci se traduit par l'ajout d'un nouvel état dans l'automate du module central (figure 4) et correspond à une nouvelle séquence d'exécution prenant en charge la libération des noeuds. Néanmoins, cette approche nous permet de conserver à la fois notre découpage initial des fonctions réalisées par les différents modules, l'indépendance totale des ordonnanceurs vis-à-vis du *Desktop computing* et la vision de haut niveau des tâches de type << meilleur effort >> . En outre, elle illustre parfaitement les facilités offertes par OAR pour l'implémentation de fonctions transversales dans l'ensemble du système.

Avec cette approche, il est possible de définir une politique de sélection des tâches à annuler, comme un tri selon leur date de démarrage (afin de stopper les tâches les plus jeunes et essayer de terminer les plus anciennes) ou selon le nombre de noeuds occupés (pour tenter de minimiser le nombre de tâches << meilleur effort >> à annuler). Une fois de plus les modifications à apporter au système sont simples et peu nombreuses. Ceci est entièrement la conséquence de la haute modularité, de l'ouverture (par la base de données) et du haut niveau (obtenu grâce à Perl) du système OAR.

5.3. Performances

Deux plates-formes ont été utilisées pour les tests de performances. La première est constituée de 5 PC (bi-processeurs Xeon 2,4Ghz 512Mo RAM, réseau ethernet 1 Gbits), par la suite elle est dénommée plate-forme Xeon. La deuxième est la plate-forme Icluster (225 processeurs PIII 733, 256Mo, réseau ethernet 100 Mbits). Lors des tests, 119 nœuds de calcul étaient disponibles. Pour l'ensemble des tests sur la plate-forme Xeon, le client test, le module central et la base de donnée pour OAR ou le démon serveur pour OpenPBS sont exécutés sur la même machine biprocesseur. Sur la plate-forme Icluster le client est exécuté sur une nœud distinct et les aspects serveurs (module central, base de donnée et démon OpenPBS) sur une machine PIII 866Mhz, 256Mo de mémoire. Les tâches soumises correspondent à la demande d'exécution d'un processus exécutant la commande système *date*, le nombre de nœuds réservés dépend de l'expérience menée. Dans les tests nous limitons le nombre maximum de tâches soumises simultanément à 100. Cette limite est largement suffisante si on considère des soumissions de tâche directement par les utilisateurs, et pour des soumissions automatiques (via des scripts ou des agents logiciels) elle reste convenable. De plus au-delà de cette limite, nous avons eu des problèmes de stabilité avec le système OpenPBS.
La figure 6 montre les performances obtenues sur la plate-forme Xeon. Le test consiste à mesurer le temps moyen de réponse pour une tâche demandant un nœud en fonction du nombre de tâches soumises simultanément. Le temps de réponse unitaire mesuré est la différence entre la date de terminaison de la tâche et la date de soumission. La variance est faible et non représentée sur le graphique. Les différentes courbes correspondent au temps obtenu par OpenPBS, puis OAR avec et sans vérification de l'état des nœuds avant l'exécution de la tâche. La vérification consiste à tester l'état du nœud et son accessibilité en effectuant une première exécution distante d'une commande *vide* via *rsh* ou *ssh*. Ainsi, la chaîne de

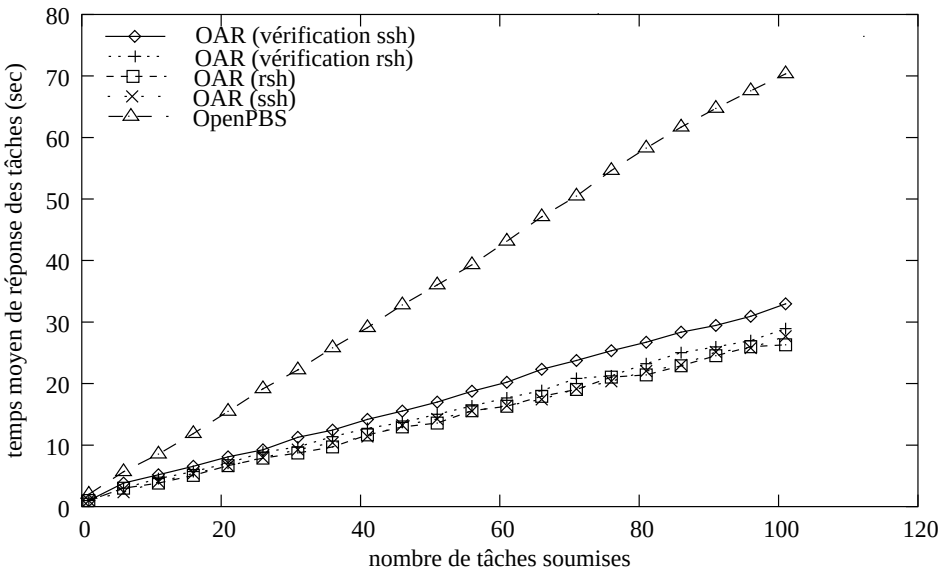

FIG. 6 – Temps de réponse moyen d'une tâche demandant 1 nœud en fonction du nombre de tâches soumises pour les systèmes OpenPBS et OAR (avec et sans vérification des états des noeuds) sur la plate-forme Xeon (4 nœuds de calcul).

lancement est testée jusqu'à la couche application.

Sur cette plate-forme récente on observe que le niveau de performance atteint par notre système est meilleur que celui obtenu avec OpenPBS. Malgré une approche de haut-niveau et l'emploi d'un langage interprété, la montée en charge pour notre système est meilleure. De plus la base de donnée, pour le traitement de 10 tâches, reçoit 350 requêtes *SQL* soit environ 70 reqs/sec sur la plate-forme Xeon. Cette charge est une très faible par rapport à la capacité de traitement de la base (>3000 reqs/sec).

Cela montre un bon emploi de la base de donnée et une bonne architecture du système. De plus, le surcoût d'une vérification systématique de l'état des nœuds est faible.

La figure 7 présente les mêmes tests que précédemment sur la plate-forme Icluster entre OpenPBS et OAR sans vérification d'état. Sur cette plate-forme d'ancienne génération, les coûts de l'emploi d'une approche de haut-niveau n'ont pas pu être amorti. Le taux de remplissage des nœuds de calcul est moins bon que celui d'OpenPBS. Nous n'avons pas détaillé les différents coûts, mais sur cette plate-forme le coût d'un lancement d'un script Perl est important puisqu'il nécessite une phase de pré-compilation (0,3 seconde pour débuter une commande de soumission). Or ce type de coût intervient au lancement de chaque module à chaque cycle du module central.

La figure 8 présente le temps de réponse moyen d'une tâche en fonction du nombre de noeuds demandés. A chaque point 20 tâches sont soumises simultanément. Le système OAR sans vérification d'état possède de bonne performance. Avec vérification d'état, le plus faible niveau de performance par rapport à OpenPBS reste convenable avec *rsh*. Mais, il est nettement moins bon avec le protocole *ssh*. Il faut noter qu'OpenPBS n'effectue pas de vérification systématique au niveau du protocole d'exécution sur l'ensemble des nœuds attribués avant le lancement de la tâche. Cette vérification systématique confère au système OAR un niveau de qualité de service élevé puisque chaque nœud alloué est testé jusqu'au niveau applicatif de lancement d'exécution à distance. En terme de coût, on retrouve les mêmes constatations que précédemment, mais ils sont atténués lorsque le nombre de ressources demandées augmentent.

Globalement, les performances sont bonnes sur des machines récentes où l'emploi d'une approche de haut-niveau n'est pas pénalisante. De plus la gestion d'un nombre important de ressources est maîtrisée.

6. Conclusion

Dans cet article, nous avons présenté un nouveau système de gestion de ressources pour grappe, baptisé OAR. La principale contribution de OAR provient de sa conception. Il repose sur deux composants de haut-niveau : une base de données SQL et un outil générique d'administration de grappes passant à

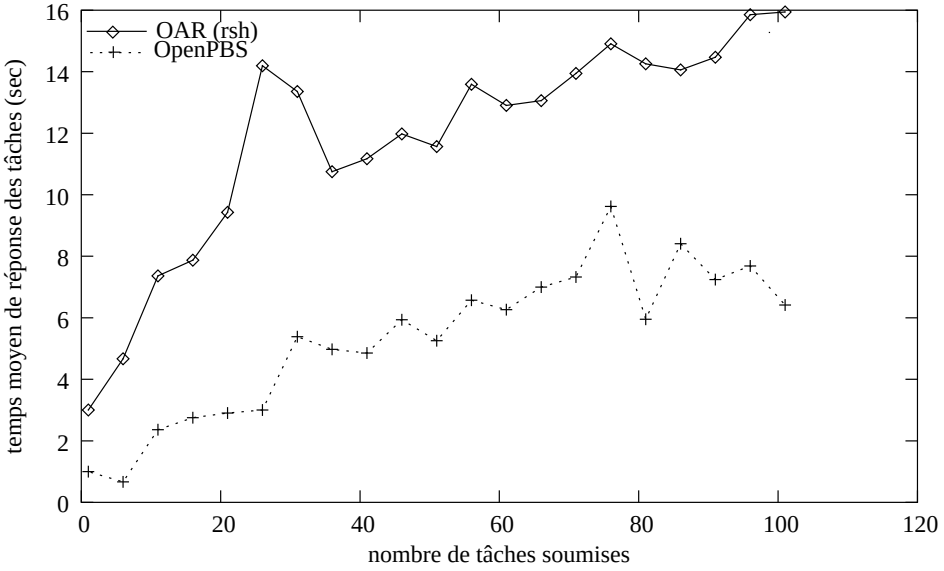

FIG. 7 – Temps de réponse moyen d'une tâche demandant 1 nœud en fonction du nombre de tâches soumises pour les systèmes OpenPBS et OAR (sans vérification des états des noeuds) sur la plate-forme Icluster (119 nœuds de calculs).

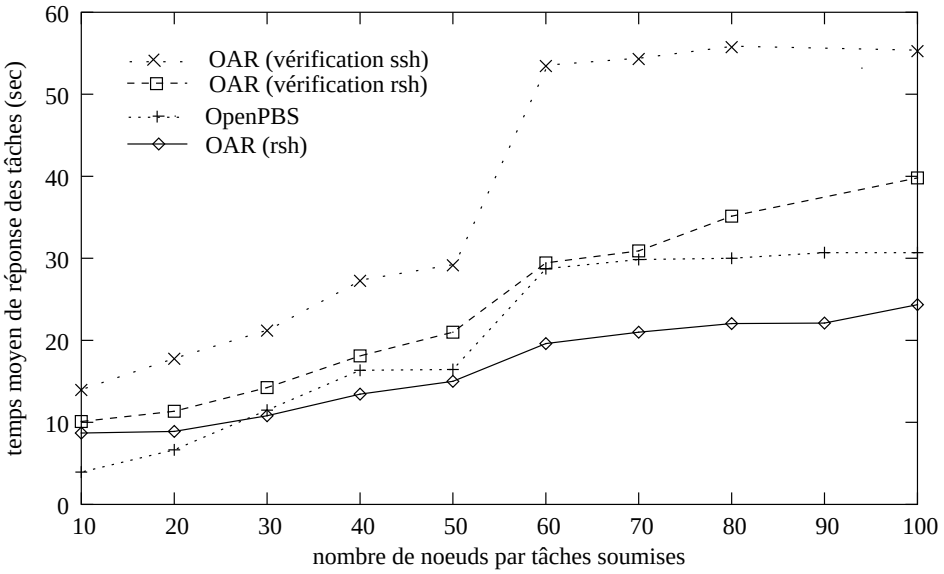

FIG. 8 – Temps de réponse moyen d'une tâche en fonction du nombre de nœuds demandés pour les systèmes OpenPBS et OAR (avec et sans vérification des états des noeuds) sur la plate-forme Icluster. Pour chaque point, 20 soumissions sont soumises simultanément.

l'échelle. La base de donnée est utilisée comme seul médium d'échange d'information entre les différents modules et assure ainsi une ouverture totale du système. La conception de l'ensemble est volontairement très modulaire et l'implémentation dans un langage de haut niveau assure au système extensibilité, robustesse et simplicité. Nous avons validé ces objectifs, d'une part en implémentant une politique d'exploitation de type *Global Computing* à partir du système initial et d'autre part en montrant le bon niveau de performance de OAR en comparaison à celui de OpenPBS. Ces résultats démontrent l'avantage de l'emploi d'une méthode moderne de conception. Le résultat est une plate-forme ouverte et performante d'expérimentation et de recherche dans laquelle nous allons pouvoir envisager d'aborder de nouvelles problématiques (ordonnancement par lots, tâches malléables, plate-formes hétérogènes, réseaux non fiables, relation avec le *Grid Computing*,...) sans nous heurter à la complexité du système de gestion de ressources sous-jacent.

Bibliographie

1. *Distribution CLIC (Cluster Linux pour le calcul).* http ://clic.mandrakesoft.com/.
2. *Maui Scheduler.* http ://www.supercluster.org/maui.
3. *Maui Scheduler Molokini Edition.* http ://mauischeduler.sourceforge.net/.
4. *MySQL Reference Manual.* http ://www.mysql.com/documentation/.
5. *OpenPBS.* http ://www.openpbs.org.
6. Carl Albing. Cray NQS : production batch for a distributed computing world. In *Proceedings of the 11th Sun User Group Conference and Exhibition*, pages 302–309, Brookline, MA, USA, December 1993. Sun User Group, Inc.
7. M. Baker, G. Fox, and H. Yau. Cluster computing review. Technical report, Northeast Parallel Architectures Center, Syracuse University, 1995.
8. Ron Brightwell and Lee Ann Fisk. Scalable parallel application launch on cplant. In *Proceedings of SC'2001.*
9. Michael Brim, Ray Flanery, Al Geist, Brian Luethke, and Stephen Scott. Cluster command and control (c3) tool suite. In *Proceedings of 3rd Workshop on Distributed and Parallel Systems in conjunction with EuroPVM/MPI*, 2000.
10. Jeff Chase, Laura Grit, David Irwin, Sara Sprenkle, and Justin Moore. Dynamic resource trading in a grid site manager. In *High Performance Distributed Computing (HPDC-12)*, 2003.
11. G. Fedak, C. Germain, V. N'eri, and F. Cappello. Xtremweb : A generic global computing system. In *In IEEE Int. Symp. on Cluster Computing and the Grid*, 2001.
12. Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. Storm : Lightning-fast resource management. In *Suppercomputing 2002*, Baltimore, MD, November 2002.
13. D. R. Ghormley, D. Petrou, S. H. Rodrigues, and A. M. Vahdat. GLUnix : A Global Layer Unix for a network of workstations. *Software Practice and Experience*, 28(9), 1998.
14. National HPCC. Software exchange review. *http ://www.crpc.rice.edu/NHSEreview*, 1996.
15. IBM Corporation. *Using and Administering LoadLeveler – Release 3.0*, 4 edition, August 1996. Document Number SC23-3989-00.
16. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.
17. Cyrille Martin and Wilfrid Billot. Lancement d'application sur des grappes de grande taille. In *proceedings of RenPar'14*, pages 17–24, 2002.
18. Cyrille Martin and Olivier Richard. Parallel launcher for cluster of pc. In *Proceedings ParCo 2001*, pages 473–480, 2001.
19. Cyrille Martin and Olivier Richard. Algorithme de vol de travail appliqué au déploiement d'applications parallèles. In *Soumis à RenPar'15*, 2003.
20. Emil Ong, Ewing Lusk, and William Gropp. Scalable unix commands for parallel processors : A high-performance implementation. In *Proceedings of Euro PVM-MPI*, 2001.
21. Douglas Thain and Miron Livny. Condor and the grid. In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, *Grid Computing : Making The Global Infrastructure a Reality.* John Wiley, 2003.
22. Songnian Zhou. LSF : load sharing in large-scale heterogeneous distributed systems. In *Proceedings*

of the Workshop on Cluster Computing, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University.